A Workbench for Experimenting with Loop Optimizations

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by

Dhanunjaya Naidu.Yandrapu

to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

December, 2004

Certificate

This is to certify that the work contained in the thesis entitled "*A Workbench for Experimenting with Loop Optimizations*", by *Dhanunjaya Naidu.Yandrapu*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

[illegible]

December, 2004

(Dr.Sanjeev Kumar.Aggarwal)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

[illegible]
8/12/04

Abstract

Parallel machines are becoming increasingly important to many compute intensive applications. Parallelizing compilers exploit the features of such architectures by restructuring the loops in the source program. One of the main difficulties for these compilers is how to combine the loop optimizations for the best performance. There are some situations in which these compilers fail to exploit the inherent parallelism present in the programs. This is due to the conservative approach taken by these compilers. In the above two situations, user intervention helps the compiler better parallelize the program. Today, commercial compilers automatically tailor programs but that tailoring may not be suitable for our purpose. Thus, application developers must either settle for lackluster performance or manually transform the code. This manual transformation is laborious and error prone. To develop parallelizing compilers with a new set of loop optimizations, the developer needs to study the effect of these optimizations. To address these problems, we have developed a tool which can be used as a platform for experimenting with different kinds of loop optimizations and studying the effect of these optimizations. It applies loop optimizations under user control.

Acknowledgements

I take this immense opportunity to express my sincere gratitude towards my supervisor Dr. Sanjeev Kumar. Aggarwal, for his invaluable guidance. It would have never been possible for me to take this project to completion without his innovative ideas and his relentless support and encouragement. I consider myself extremely fortunate to have had a chance to work under his supervision. Inspite of his hectic schedule he was always approachable and took his time off to attend to my problems and give the appropriate advice. It has been a very enlightening and enjoyable experience to work under him.

I also wish to thank whole heartily all the faculty members of the Department of Computer Science and Engineering for the invaluable knowledge they have imparted to me and for teaching the principles in most exciting and enjoyable way. I also extend my thanks to the technical staff of the department for maintaining an excellent working facility.

I wish to thank my friends B.R.P. Rohini Kumar, S.C.G. Kiran Babu and K.P.V. Papa Rao for their guidance and encouragement throughout my stay at IITK. I also wish to thank P. Lova Raju and Vinoth.B.R. for their valuable suggestions during this thesis writeup.

Finally, I thank my mother and my brothers for always being there for me, for bringing me to this point in life.

Contents

List of Figures

Chapter 1

Introduction

Parallel machines are becoming increasingly important to many compute intensive applications. In order to exploit the features of such architectures, sophisticated compilers are needed. The compilers need to find potential parallelism within the sequential program and generate parallel code for these machines. Compilers generally use statement reordering to obtain potential parallelism. In a typical program 80% of the execution time is spent around 20% of the code. Loops are such small pieces of code in which programs spend most of their execution time. By properly restructuring the loops and generating efficient code most of the execution time can be cut down.

1.1 Motivation

Many loop optimization techniques for *Fortran do* loops have been proposed [1] [2]. These techniques improve the performance either by reordering the statements of the body of the loop or by reordering the iterations of the loop. Parallelizing compilers apply some of these optimizations for improved performance. One of the main difficulties for these compilers is how to combine the loop optimizations for the best performance. There has been no general frame work that combines all the loop optimizations. It is still an area of research. There are some situations in which these compilers fail to exploit the inherent parallelism present in the programs. This is due to the conservative approach taken by these compilers. In the above two situations, user intervention helps the compiler better parallelize the program. To develop parallelizing compilers with a new set of loop optimizations, the developer needs to study the effect of these optimizations. A tool that applies loop optimizations under user control is very helpful in this case.

Today, commercial compilers automatically tailor programs but that tailoring may not be suitable for our purpose. Thus, application developers must settle for lackluster performance or manually transform the code. This manual transformation is laborious and error prone. These situations necessitates a tool that applies loop optimizations under user control.

1.2 Problem Statement

Our goal is to develop a benchmark which can be used as a platform for experimenting with different kinds of loop optimizations and studying the effect of these optimizations. It applies loop optimizations under user control, i.e., the user specifies what optimization to be applied. The tool accepts *C* language programs as the input. It produces the optimized code in *C* language. The user interaction with the tool is through a graphical user interface (GUI).

1.3 Our Approach

We transform the source program into intermediate representation (IR). The IR preserves high level information such as loops, conditionals etc. The IR is then analyzed for applying the specified loop optimization. If the application of the particular optimization does not change the semantics, then the optimization is applied. The IR is finally transformed back to source language.

The key issues that are to be handled are:

1. Generating the IR from the source program

2. Analyzing the IR to determine whether the loop optimization can be applied

3. Generating the new IR corresponding to the new loop optimization

4. Generating source code from the IR

1.4 Related Work

1.4.1 Loop Tool

Loop tool [3] provides an integrated collection of optimizing transformations and provides users with precise control over how these optimizations should be applied. It is a source-to-source transformer accepting *Fortran 77* programs. To improve performance, developers augment their code with directives to control the application of optimizing transformations. It can apply *loop fusion*, *unroll-and-jam*, *multi-level blocking* and *iteration space splitting*. When the tool is run on a program augmented with directives, it checks the legality of the optimization specified by the directives, performs the optimization and outputs the optimized code in source form.

1.4.2 The LAMBDA Loop Transformation Toolkit

The LAMBDA loop transformation toolkit [4] is an implementation of the non-singular matrix transformation theory, which can represent any linear one-to-one transformation. The LAMBDA's interface is independent of any compiler intermediate representation (IR), i.e., all the modules of the toolkit work on its own representation. The toolkit provides a set of routines for applying loop transformations. It requires the specification of loops in LAMBDA's format for applying the loop transformations. It produces the output in LAMBDA's representation.

1.5 Organization of the Report

The rest of this report is organized in the following lines: chapter 2 describes data dependence analysis and various data dependence tests that have been proposed in the literature. Chapter 3 describes the loop optimization techniques and the preconditions under which they can be applied. Chapter 4 describes the design and implementation of the workbench. Finally, we conclude in chapter 5 giving directions for future work. Appendix gives the source code organization.

Chapter 2

Dependence Analysis

Data dependence relations are used by compilers to represent the essential ordering constraints among statements in a program. These relations allow the compiler more flexibility in reordering the statements for improved performance. In this chapter we describe notation for data dependence relations and how they arise in loops.

2.1 Criteria for Program Optimization

Program optimization means transforming the original program into a semantically equivalent program that is expected to be time or space efficient than the original program.

The first criterian for code optimization is preservation of semantics of the program, i.e., the translated program must compute the same thing as the original program does. It must not be the case that the transformed code produces some error or some incorrect computation which would not have happened before transformation.

Sometimes a transformation may change the computation slightly but that change must be acceptable. For example, a divide by zero error at some point in the original program might occur at a different point in the transformed program because of code motion. Though it may complicate the debugging process, that change is acceptable.

Secondly, the transformed program should produce some improvement over the original program. Improvement may be either in time (time efficient) or memory.

2.2 Dependence Analysis

Compilers need to reorder the statements and the iterations of the loops for generating efficient code for parallel machines. Some statements have constraints on the order of their execution. All those ordering constraints must be preserved in the reordered program, otherwise the meaning of the program will change. Dependence analysis determines all those ordering constraints.

Dependence is a relationship between two computations that places constraints on their execution order. Dependence analysis identifies these constraints, which are then used to determine whether a particular transformation can be applied without changing the semantics of the computation. There are two kinds of dependences

1. Control dependence
2. Data dependence

2.2.1 Control Dependence

There exists a control dependence from statement S_1 to statement S_2 if S_1 determines whether S_2 is to be executed or not.

```
S1:   IF ( X < 10 ) THEN
S2:       B = X
      ENDIF
```

In the above example, execution of statement S_2 is dependent on statement S_1. So there is a control dependence from S_1 to S_2.

2.2.2 Data Dependence

Two statements have a data dependence if they cannot be reordered due to conflicting uses of the same variable. For example

S_1: X = 100
S_2: Y = 200
S_3: Z = X + Y

The values of X and Y used in S_3 are the values assigned in S_1 and S_2 respectively. S_3 cannot be executed until S_1 and S_2 have completed their execution. But there is no dependence between S_1 and S_2, i.e., they can be executed in any order.

There are three types of data dependences:

1. Flow dependence: Flow dependence occurs when a variable is assigned or defined in one statement and it is used in subsequently executed statements. For example,

   S_1: X = 10
   S_2: Y = X + 20

   Here the value of X used in S_2 is the value defined in S_1. So there is flow dependence between S_1 and S_2 (more precisely, S_2 is flow dependent on S_1).

2. Anti-dependence: Anti-dependence occurs when a variable is used in one statement and it is reassigned in a subsequently executed statement. For example,

   S_1: Y = X + 20
   S_2: X = 10

3. Output-dependence: Output-dependence occurs when a variable is assigned in one statement and it is reassigned in a subsequently executed statement. For example,

```
      DO I= 1, N
$S_1$ :   A(I) = ···
$S_2$ :   ··· = A(I) + c
$S_3$ :   A(I-1) = ···
      ENDDO
```

Figure 2.1: Data dependence in a loop

S_1: X = Y + 20

S_2: X = 10

The last two dependences occur because of reuse of the variables. This dependence can be eliminated if we introduce new variables and replace the old variable by the new variables in all the subsequent statements. These dependences (anti and output) are referred to as false dependences.

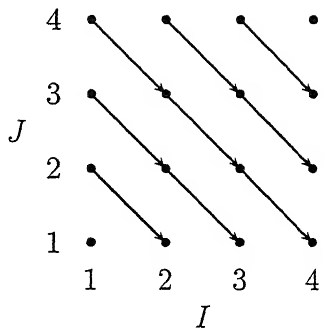
2.3 Data Dependence in Loops

In straight-line code, each statement is executed at most once. But in case of loops, a statement in a loop may be executed more than once. Dependence can flow from an instance of a statement in one iteration to an instance of a statement in another iteration (possibly to the same statement instance). Dependences between two different iterations are said to be loop carried dependences and dependences between statements of the same iteration are said to be loop independent.

For example, consider the loop in figure 2.1. There is a loop independent dependence from statement S_1 to statement S_2, because $A(I)$ is defined in the statement S_1 and it is used in the statement S_2 of the same iteration. There is a loop carried dependence from S_1 to S_3. The value of $A(I)$ defined in i^{th} iteration is defined again in the $(i+1)^{th}$ iteration.

```
DO I= 1, 4
  DO J= 1, 4
    A(I,J) = B(I,J)+k
    B(I,J) = A(I-1,J+1)-C(I,J)
  ENDDO
ENDDO
```

(a) Loop nest



(b) Iteration space dependence graph

Figure 2.2: Iteration space example

2.3.1 Iteration Space

The dependences between the statement instances in a loop are represented using iteration space dependence graph. Iteration space consists of one point for each iteration of the loop (possibly a nested loop). A loop carried dependence from iteration i to iteration j is represented by an arc from the point corresponding to iteration i to point corresponding to iteration j. The iteration space dependence graph for the loop in figure 2.2(a) is shown in figure 2.2(b).

2.3.2 Dependence Distance

Each point in the iteration space of a nested loop of depth d is represented by a vector $V = (I_1, I_2, \cdots, I_d)$, where $I_1, I_2 \cdots I_d$ are the index variables of the loop nest. This vector V is called Iteration vector. Dependence distance from source iteration i to destination iteration j is computed by computing the vector difference between the iteration vectors corresponding to iterations j and i. For example, the dependence distance from $(2,3)$ to $(3,2)$ is $(1,-1)$.

2.4 Subscript Analysis

Subscript analysis determines when two array references refer to the same element in different iterations. The compiler applies various tests on the subscript expressions to determine dependences between them. If some dependences are found, the compiler represents them using either dependence distance vectors or direction vectors. If the subscript expressions are too complex to be analyzed (for example, non linear subscript expressions), the compiler may assume that there is a dependence.

The first step in analyzing dependences between the array references is to set up dependence equations. The index variables are the unknowns in the equation. The unknowns may have constant coefficients and the equation may contain a constant term. Once dependence equations are found, the compiler might impose certain constraints on the unknowns (constraints on the index variables). The dependence equations together with the constraints form a dependence system. The dependence system consists of a set of linear equality and inequality equations. These equations are then solved for integer solutions. If there are no integer solutions for the dependence system, the references are independent.

```
DO I= 1, 100
   DO J= 2, 50
      A(I,J) = B(I,J) - C(I,J)
      D(I,J) = A(I-2, J)*k
   ENDDO
ENDDO
```

For example, the dependence equations for the references to array A in the loop shown above are

$$i_d + 1 = i_u - 1$$

$$j_d + 2 = j_u + 2$$

$$0 \leq i_u \leq 99$$

$$0 \leq i_d \leq 99$$

$$0 \leq j_u \leq 48$$

$$0 \leq j_d \leq 48$$

The two equality dependence equations correspond to the two dimensions of the array A. As the equations have integer solutions, the two array references have a flow dependence.

2.5 Data Dependence Tests

Data dependence tests determine when two array references refer to the same array element. If the array is multidimensional, it is same as determining integer solutions to a set of linear equations. Many dependence tests have been proposed. They all differ in accuracy and efficiency. All these tests are conservative, meaning that they never report non-existence of a dependence when there is actually a dependence but they may report a dependence even though there is no dependence. Some of the widely used data dependence tests are discussed in the following sections.

2.5.1 The GCD Test

The GCD test [5] is the most basic of the dependence tests. This test is generally used in other dependence tests to disprove a dependence. It is very simple and inexpensive. It is based on an elementary number theorem which states that a linear equation has integer solutions if and only if the greatest common divisor of the coefficients on the left hand side of the equation evenly divides the coefficient on the right hand side of the equation. The GCD test uses this theorem to report dependences. If the integer divisibility is not satisfied, the GCD test reports that there exists no dependence. Otherwise, it reports that there is a dependence.

The GCD test is limited in many respects. Firstly, it can only consider a single dimension of multidimensional array references at a time. Secondly, if any of the coefficients of the equation on the left hand side equals one, it always reports a dependence. Lastly, it does not take any inequalities into consideration.

2.5.2 The Banerjee Test

The banerjee test [5] is one of the widely used dependence tests. It calculates the minimum and maximum values that the expression on the left hand side of the equation can achieve based on the bounds of the variables involved in the equation. If the constant value on the right hand side of the equation does not fall within this range, the test reports that there does not exist a dependence. If it does fall within the range, it returns a maybe answer because the equation may have only real solutions but no integer solutions. Like GCD test, it also considers single dimension of a multidimensional array at a time.

2.5.3 The Omega Test

The omega test [6] is an exact test. It determines the existence of integer solutions to a set of linear equalities and linear inequalities. It first converts the system of linear equalities and inequalities into a system involving only linear inequalities. It applies bound normalization and GCD test to determine if the system is inconsistent. If it is inconsistent, it reports that no integer solutions exist to the system of linear equations. Otherwise, it uses an extension to Fourier Motzkin Variable Elimination (FMVE) to determine integer solutions. The FMVE determines the existence of real solutions. It determines the solutions by eliminating variables. Intuitively, the elimination of a variable may be viewed as projecting an n-dimensional polyhedron onto an *(n-1)*-dimensional surface. If the *real shadow* contains no integers, then the original object contains no integers, the omega test reports that there does not exist any dependence. However, if the real shadow has integer solutions, the original object may or may not actually have integer solutions. The omega test handles this case by defining a subset of the *real shadow* called the *dark shadow* corresponding to the area of the object where integer solutions definitely exists. If the *dark shadow* contains integer solutions, the omega test reports that there exists integer solutions. If the *dark shadow* contains no solutions and the *real shadow* contains integer solutions, then it recursively searches the solution space until it disproves or a solution is found.

Chapter 3

Loop Optimizations

In this chapter we describe loop restructuring and loop reordering transformations. Loop reordering transformations improve performance by changing the order of execution of the iterations of the loop nest and hence these transforms are not always legal. Loop restructuring transformations on the other hand do not change the order but change the body of the loops.

3.1 Loop Unswitching

Loop unswitching [7] avoids the evaluation of loop independent conditional expression inside a loop. If a conditional occurs inside a loop and its conditional expression is loop invariant, then this transformation moves the conditional out of the loop.

For example, if the condition in figure 3.1(a) is loop independent, then the if-then-else conditional can be moved out of the loop. The transformation results in the loop shown in figure 3.1(b).

This optimization has the advantage of saving the overhead of a conditional branch inside a loop. The disadvantage of this optimization is that more instruction space is required because the body is replicated twice.

```
DO I= 1, N
  IF(condition) THEN
    A(I) = B(I)+C(I)
  ELSE
    A(I) = B(I)-C(I)
  C(I) = C(I)+A(I)
ENDDO
```

(a) Original loop

```
IF (condition) THEN
  DO I=1, N
    A(I) = B(I)+C(I)
    C(I) = C(I)+A(I)
  ENDDO
ELSE
  DO I=1, N
    A(I) = B(I)-C(I)
    C(I) = C(I)+A(I)
  ENDDO
```

(b) After unswitching

Figure 3.1: Loop unswitching

3.2 Loop Unrolling

The original motivation for this transformation is to reduce the loop overhead (i.e., to reduce the number of checking conditions) [7, 8]. This technique is very useful for generating long instruction words for VLIW architectures. In this transformation, the body of the loop is replicated (unrolled) for improved performance. Unrolling factor is the number of times the body is unrolled. Deciding appropriate value for the unrolling factor [9] has an impact on the performance. Typically, the value of the unrolling factor depends on the cache size of the machine. Unrolling the loop shown in figure 3.2(a) by 2, results in the loop shown in figure 3.2(b).

```
DO I = 1, 100
  A(I) = B(I)+C(I)
ENDDO
```

(a) Before unrolling

```
DO I = 1, 100, 2
  A(I) = B(I)+C(I)
  A(I+1) = B(I+1)+C(I+1)
ENDDO
```

(b) After unrolling

Figure 3.2: Loop unrolling

```
DO I = 1, 96, 6
  A(I) = B(I)+C(I)
  A(I+1) = B(I+1)+C(I+1)
  A(I+2) = B(I+2)+C(I+2)
  A(I+3) = B(I+3)+C(I+3)
  A(I+4) = B(I+4)+C(I+4)
  A(I+5) = B(I+5)+C(I+5)
  A(I+6) = B(I+6)+C(I+6)
ENDDO

DO I = 97, 100
  A(I) = B(I)+C(I)
ENDDO
```

Figure 3.3: Unrolling example

Care must be taken when the number of times the loop executed is not a multiple of the unrolling factor. In such a case, code for executing the last iterations of the loop needs to be provided along with the unrolled loop. For example, if the loop in figure 3.2(a) is unrolled by 6, code for executing the last 4 iterations of the loop has to be provided after the unrolled loop as shown in figure 3.3.

Loop unrolling improves the performance by

- Reducing the number of loop testing conditions
- Increasing the size of the loop body, which results in more number of instructions getting exposed for parallelism (ILP)
- Improving register locality.

All three of these improvements are shown in an example shown in the figure 3.4. Loop testing conditions are reduced by half because two iterations are performed before test and branch. Instruction parallelism is increased because the second assignment can be performed while the results of the first assignment are being stored. If array elements are assigned to registers, register locality will improve because $a[i]$ and $a[i+1]$ are used twice in the unrolled loop body.

```
DO I = 1, 64
  A(I) = A(I)+A(I-1)*A(I+1)
ENDDO
```

(a) Original loop

```
DO I = 1, 64, 2
  A(I) = A(I)+A(I-1)*A(I+1)
  A(I+1) = A(I+1)+A(I)*A(I+2)
ENDDO
```

(b) Unrolled loop

Figure 3.4: Advantages of loop unrolling

The disadvantage of this optimization is that it requires more instruction space because the body is replicated many number of times. Excessive unrolling of loops also has negative effect. If the working set of the unrolled loop body exceeds the number of registers available, some of the registers have to be reallocated which is an overhead. Since this transformation does not change the order of execution, it is always legal.

3.3 Loop Interchanging

Loop interchanging [10] is an important loop reordering transformations. It switches the position of two loops in a perfectly nested loop. This transformation is used to parallelize the innermost loop [11]. Parallelization/Vectorization is not possible if the innermost loop has dependences between iterations. Interchanging can make vectorization possible by moving the dependent innermost loop with the independent outer loop.

For example, consider the matrix multiplication shown in figure 3.5(a). The inner loop is not vectorizable because the values of C are from different rows. If the J^{th} and K^{th} loops are interchanged as shown in figure 3.5(b), the inner loop can be vectorized by considering $B(I, K)$ as a scalar.

Loop interchanging can be used to improve the performance in many ways. For example, if the outer loop iterates many times and the inner loop iterates only a few times, there will be a significant loop startup overhead for the inner loop. If we

```
DO I = 1, N
  DO J = 1, N
    DO K = 1, N
      A(I,J) += B(I,K) * C(K,J)
    ENDDO
  ENDDO
ENDDO
```

(a) Before interchanging

```
DO I = 1, N
  DO K = 1, N
    DO J = 1, N
      A(I,J) += B(I,K) * C(K,J)
    ENDDO
  ENDDO
ENDDO
```

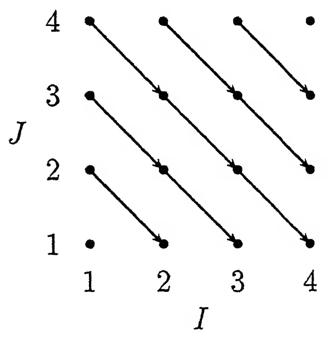
(b) After interchanging

Figure 3.5: Loop interchanging

interchange these loops then the startup overhead gets reduced. This transform can also improve data cache performance by changing the stride of array accesses in the inner loop.

```
DO I = 1, 4
  DO J = 1, 4
    B(I,J) = C(I,J)-k;
    A (I,J) = B(I-1,J+1)+k
  ENDDO
ENDDO
```

(a) Original loop



(b) Iteration space

Figure 3.6: Loop with dependence distance $(1, -1)$

Since interchanging changes the order of execution, this transformation is not always legal. For example, consider the loop shown in figure 3.6(a). It has loop carried dependences with distance $(1, -1)$ as shown in figure 3.6(b). If the loop is interchanged, $B(2, 1)$ gets used before $B(1, 2)$ is defined. Obviously, this will change the semantics of the program. Hence, such a transformation is illegal.

Interchanging of two loops is legal if all the dependences before interchanging are also preserved after interchanging. In general, if two loops p and q in a perfect loop nest of d loops are interchanged, each dependence vector $V = (v_1, \cdots, v_p, \cdots, v_q, \cdots, v_d)$ in the original loop nest becomes $V^* = (v_1, \cdots, v_q, \cdots, v_p, \cdots, v_d)$ in the transformed loop. If V^* is lexicographically positive (i.e., the first non zero element in the dependence vector is positive) [1], this transformation is legal.

Chapter 4

Design and Implementation

This chapter describes the design and implementation of our tool. It is a GUI based tool. It takes the *C* language source program as the input, displays the loops in the program to the user and interacts with the user to apply various kinds of loop optimizations on the *for* loops. Finally, it generates optimized code in *C* language.

The tool includes some of the phases of the traditional compiler and some extra phases that are missing in a non optimizing compiler. The phases of the tool are shown in figure 4.1. It has no semantic analysis phase. It does not check to see whether a variable is declared before being used or a *continue* statement occurs within an iterative statement. It accepts syntactically and semantically correct *C* language source program, finds *for* loops in the source program and applies any one of the three loop optimizations (loop unswitching, loop unrolling and loop interchanging) on the *for* loops. In other words, the use of lexical analysis and syntax analysis phases in the compiler is just to obtain the intermediate representation (abstract syntax tree) of the source program. Each construct in the language such as statements, expressions, functions etc., are represented by a node in the abstract syntax tree. All the loop optimizations are applied on this tree represented source program. A loop optimization is applied by modifying the *for* node of the abstract syntax tree. The code generation phase generates the *C* code from the abstract syntax tree. In addition to these phases, the compiler includes a Graphical User Interface (GUI). The GUI allows the user to view the loops in the source program and choose the desired optimization to apply on a particular loop.

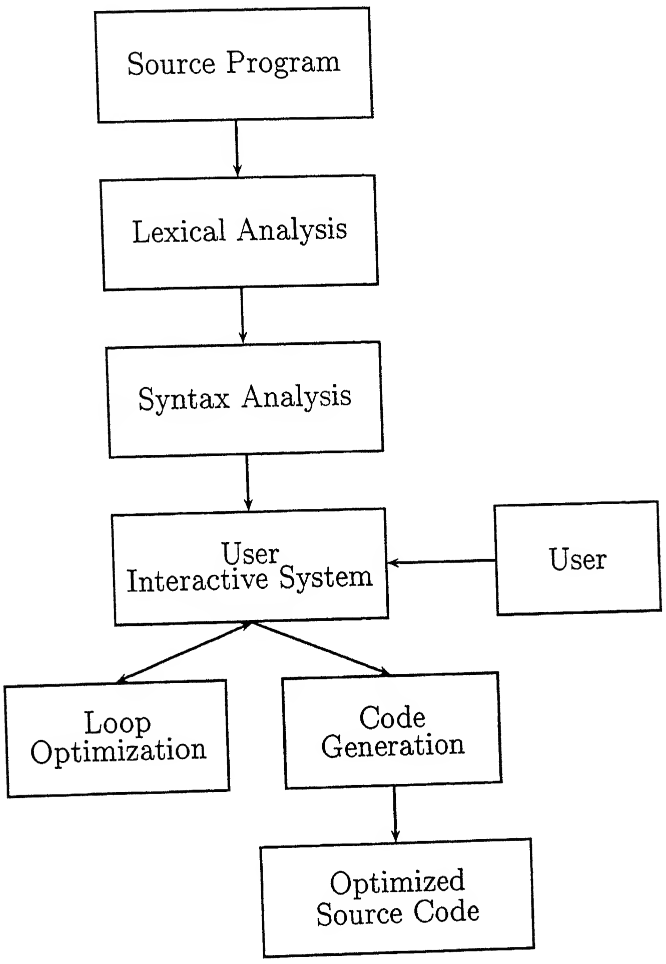


Figure 4.1: Phases of the tool

A detailed design of the tool is shown in figure 4.2. It shows the data flow between each component of the tool.

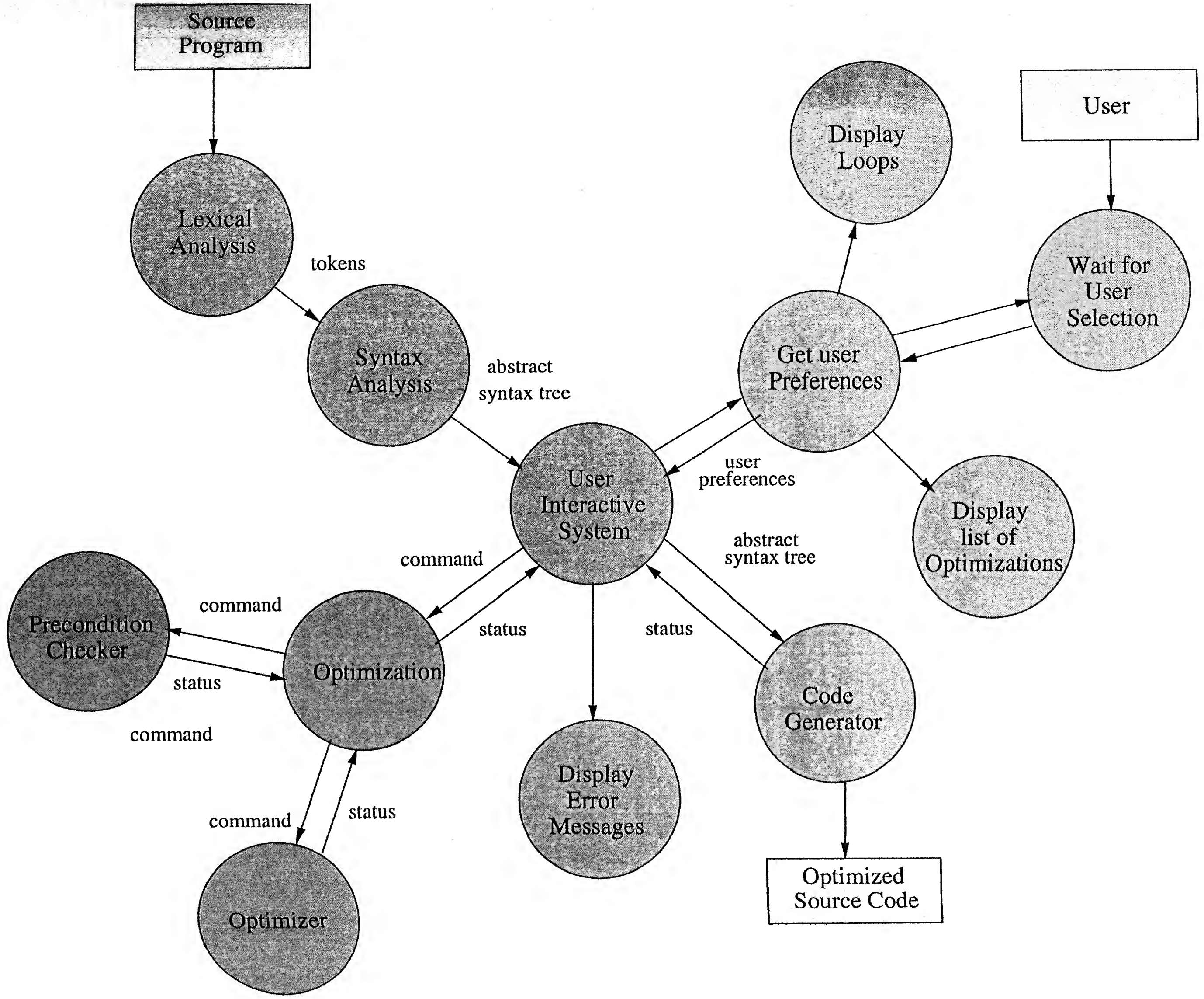


Figure 4.2: Detailed design of the tool

4.1 Lexical Analysis

Lexical analysis [12] is the process of taking a stream of characters and producing a stream of symbols called lexical tokens. White space and comments are discarded in the lexical analysis phase. This discarding simplifies the syntax analysis phase, otherwise the syntax analyzer would need to provide special rules to handle whitespace and comments.

4.1.1 Lexical Tokens

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming language. The tokens are classified into a set of token types. For example, shown below is some of the token types of the *C* language.

Token type	**Lexeme**
ID	max, value, small
NUM	25, 3.14159
LITERAL	"hello"
IF	if
FOR	for
PLUS	+
COMMA	,

The tokens for the lexemes *max*, *value* and *small* are all classified into the token type identifier (ID). For some tokens such as the reserved words *if*, *while*, *for* etc., the lexeme and the token type is the same.

4.1.2 Regular Expressions

Regular expressions [13] are the notation for specifying the lexical rules of a language. A regular expression denotes a set of strings over an alphabet. For example, the regular expressions for some of the tokens of the *C* language are shown below.

Token type	Regular expression (pattern)
ID	[a-zA-Z][a-zA-Z0-9]*
FOR	for
PLUS	+
COMMA	,

4.1.3 Lex: A Lexical Analyzer Generator

Lex [14] is a software tool that automatically generates a lexical analyzer from the lexical specification. Lexical specification program contains three parts:

Declarations
%%
Lex definitions
%%
Auxiliary procedures

1. The declaration part: It contains *C* language declarations, declarations of constants, regular expression abbreviations etc.

2. The lex definitions part: It contains lex definitions of the form

$$\begin{array}{ll} p_1 & \{action_1\} \\ p_2 & \{action_2\} \\ & \vdots \\ p_n & \{action_n\} \end{array}$$

   where each p_i is a regular expression specifying the rule for identifying a token. The $action_i$ is executed if a lexeme in the source program matches the regular expression p_i. The action part generally contains code for returning the corresponding token and its semantic value.

3. The auxiliary procedure part: This part may contain any extra routines that may be needed for lexical analysis. For example, this part may include some error routines to notify lexical errors.

Every time a regular expression matches with the input, the matched input text is stored in the global variable *yytext* and its length is specified by the global variable *yyleng*. The semantic values are communicated to the parser through the global variable *yylval*.

4.2 Syntax Analysis

Syntax analysis recognizes the structure of the programming language. It determines whether a sentence is well formed or not. Every programming language has syntax rules that describe the structure of the programs in that language. Context free grammar is a formalism which can be used to describe the syntax of most programming language constructs (statements, expressions etc.,).

4.2.1 Context Free Grammars

A context free grammar (CFG) [13] is a finite set of symbols each of which represents a language. Formally, a CFG is a 4-tuple denoted by G = (V,T,P,S), where V is finite set of variables, T is a finite set of terminals, P is a set of productions of the form $A \longrightarrow \alpha$, where A is a variable and α is a string of variables and terminals and S is the start symbol. The language generated by a CFG is called a context free language.

4.2.2 Yacc: The Parser Generator

Yacc [15] is a tool that automatically generates a LALR [12] parser (written in C) from a Yacc specification program. A Yacc specification program consists of three parts:

```
declarations
%%
translation rules
%%
supporting C routines
```

1. The declaration part: It has two optional sections. The first section consists of declarations delimited by %{ and %}. The declarations may be any function declarations that are used in the translation rules or they may be preprocessor directives. The second section consists of declarations of terminal and non-terminal symbols of the grammar. A terminal symbol is declared by using the statement

   %token ID

   All the symbols that do not have any external declarations as above are considered to be non-terminals.

2. The translation rules part: This part consists of a set of translation rules. A translation rule is a production of a context free grammar associated with a semantic action. The semantic action is executed whenever the right hand side of the production is replaced by the left hand side of the production. The semantic action usually consists of function calls to build the nodes in the parse tree.

3. The supporting C routine part: This part consists of any auxiliary procedures that are used in the semantic actions. It also consists of error routines to report parsing errors.

4.3 Abstract Syntax

Syntax analysis recognizes whether a sentence belongs to the language of the grammar. But this recognition in itself is not sufficient. The source program needs to be represented in a convenient form for further analysis. Abstract syntax [16] is a representation of the source program which is independent of the machine and the source language structure.

For example, the data structure for representing the statements of the *C* language in the abstract syntax might look as shown below:

```
struct A_stmt_ {
    stmtEnum_t kind;
    union {
        A_exp exp; /* expression statement */
        struct {A_pos pos; A_stmt exp1; A_stmt exp2; A_exp exp3;
                A_stmt stmt;} forr; /* for */
        struct {A_exp exp; A_stmt stmt;} whilee; /* while */
        struct {A_exp exp; A_stmt stmt;} dowhile; /* do-while */
        struct {A_decList declist; A_stmtList stmtlist;} compound; /* compound */
        struct {A_exp exp; A_stmt stmt;} iff; /* if */
        struct {A_exp exp; A_stmt truestmt; A_stmt falsestmt;} ifelse; /* if-else */
        struct {A_exp exp; A_stmt stmt;} switchh; /* switch */
        struct {S_symbol id; A_stmt stmt;} idlabel; /* labeled stmt */
        struct {A_exp exp; A_stmt stmt;} caselabel; /* case label */
        A_stmt defaultlabel; /* default label */
        S_symbol gotoo; /* goto */
        A_exp retexp; /* return */
    } u;
};
```

The abstract syntax for the statements of the *C* language might then look as shown below:

```
A_stmt A_expStmt(A_exp exp);
A_stmt A_forStmt(A_pos pos, A_stmt exp1, A_stmt exp2,
    A_exp exp3, A_stmt s);
A_stmt A_compoundStmt(A_decList declist, A_stmtList stmtlist);
A_stmt A_whileStmt(A_exp exp, A_stmt stmt);
A_stmt A_dowhileStmt(A_exp exp, A_stmt stmt);
A_stmt A_ifStmt(A_exp exp, A_stmt stmt);
A_stmt A_ifelseStmt(A_exp exp, A_stmt tstmt, A_stmt fstmt);
A_stmt A_switchStmt(A_exp exp, A_stmt stmt);
A_stmt A_idLabelStmt(S_symbol id, A_stmt stmt);
A_stmt A_caseLabelStmt(A_exp exp, A_stmt stmt);
A_stmt A_defaultLabelStmt(A_stmt stmt);
A_stmt A_gotoStmt(S_symbol lab);
A_stmt A_retexpStmt(A_exp exp);
A_stmt A_continueStmt();
A_stmt A_breakStmt();
A_stmt A_returnStmt();
```

The abstract syntax rules can be inserted in the semantic actions of the Yacc specification program to build a tree representation of the source program called abstract syntax tree (AST). Unlike parse tree which contains all the punctuation symbols as the leaves of the tree, abstract syntax tree does not contain unnecessary information which is used only for parsing. The abstract syntax tree represents the source program in a convenient way for further processing. The rest of the phases, the optimization phase and the code generation phase, work on the abstract syntax tree representation.

4.4 Loop Optimization

The loop optimization phase applies the three optimizations described in chapter 3. Each optimization is applied by modifying the *for* node in the abstract syntax tree

(AST). The loop optimization phase is further divided into two sub phases, the precondition checking phase and the optimization phase. The precondition checking phase ensures that the *for* loop to be optimized conforms to certain conditions (applying loop optimizations to an arbitrary *for* loop is difficult) and the loop satisfies the preconditions for the particular optimization to be applied (violation changes the semantics). The optimization phase performs the actual optimizations once the preconditions have been satisfied.

4.4.1 Precondition Checker

So far, loop optimization techniques for the *for* loops have been found. Optimizing *while* loops is considered difficult and finding such techniques is an area of research. In *C* language the *for* loop is very general, so we cannot optimize all kinds of *for* loops. For an optimization to apply on a *for* loop of the *C* language, the loop must satisfy certain constraints. We have considered *for* loops that have an index variable taking an initial value and has a step value (increment) as candidates for loop optimization.

The precondition checker first checks whether the loop is a candidate for optimization. If it is a candidate, it determines whether the particular optimization can be applied on the loop without changing the semantics of the program (see section 3.1, section 3.2 and section 3.3 for preconditions). For example, moving a conditional, whose conditional expression is not independent, out of the loop changes the semantics of the program and such transformations are illegal. The role of precondition checker for each optimization is given below:

- Precondition checking for loop unswitching involves determining whether the conditional expression of the conditional is loop invariant. If it is independent, the optimizer proceeds by applying the optimization, otherwise, the optimization is not performed. For determining whether the conditional is independent, it checks whether any of the variables in the conditional expression is modified in the loop. A variable v is modified when the variable appears on the lefthand side of an assignment expression such as $v = rvalue$ or when an increment or a decrement operator is applied to the variable ($++v$, $--v$, $v++$, and $v--$).

- The loop unrolling requires no preconditions checking; it can be applied to all the candidate *for* loops. The precondition checker checks whether the loop has an index variable with initial and final values. It also checks whether the loop has fixed increment expression.

- Precondition checking for the loop interchanging involves determining when two array references refer to the same element and if they refer to the same element, whether interchanging does not change the semantics. The precondition checking is done by establishing the dependence equations for the array references and then solving for integer solutions. If the dependence system has no integer solutions, the optimization can be applied.

We have used omega library [17] for determining the integer solutions for the dependence equations. The omega library is a set of C++ classes for manipulating integer tuple relations and sets. It allows to create relations and a set of atomic constraints on the relations. We found integer solutions by creating a relation with input variables equal to the number of unknowns in the dependence equations. The equality and inequality constraints in the dependence system are expressed as the atomic constraints of the relation. We have simplified the relation to see if there are no integer solutions. If the relation is empty, then there are no solutions to the dependence equations. Otherwise, the dependence equations have integer solutions.

4.4.2 Optimizer

The optimizer actually performs the loop optimizations. It does not check for any preconditions, which is the role of the precondition checker. Each loop optimization has its own optimizer. For example, the optimizer for loop unswitching (see section 3.1) moves the first independent conditional out of the loop, the optimizer for loop unrolling (see section 3.2) duplicates the body of the loop and the optimizer for the loop interchanging (see section 3.3) switches the position of the two loops in the loop nest.

4.5 Code Generation

The code generation phase takes the syntax tree represented form of the optimized source program (*C* language program) and translates it into its equivalent program in the *C* language. It walks through all the nodes of the abstract syntax tree, translating each of its nodes into its equivalent code in the *C* language. For example, a statement node such as

```
A_forStmt(
    A_expStmt(
        A_boperatorExp(
            A_assign,
            A_idExp(S_Symbol("i")),
            A_constExp(S_Symbol("1"))
        )
    ),
    A_expStmt(
        A_boperatorExp(
            A_le,
            A_idExp(S_Symbol("i")),
            A_idExp(S_Symbol("N"))
        )
    ),
    A_postIncExp(
        A_idExp(S_Symbol("i"))
    ),
    A_expStmt(
        A_boperatorExp(
            A_addAssign,
            A_idExp(S_Symbol("sum")),
            A_idExp(S_Symbol("i"))
        )
        )
    )
```

in the abstract syntax tree would be translated to

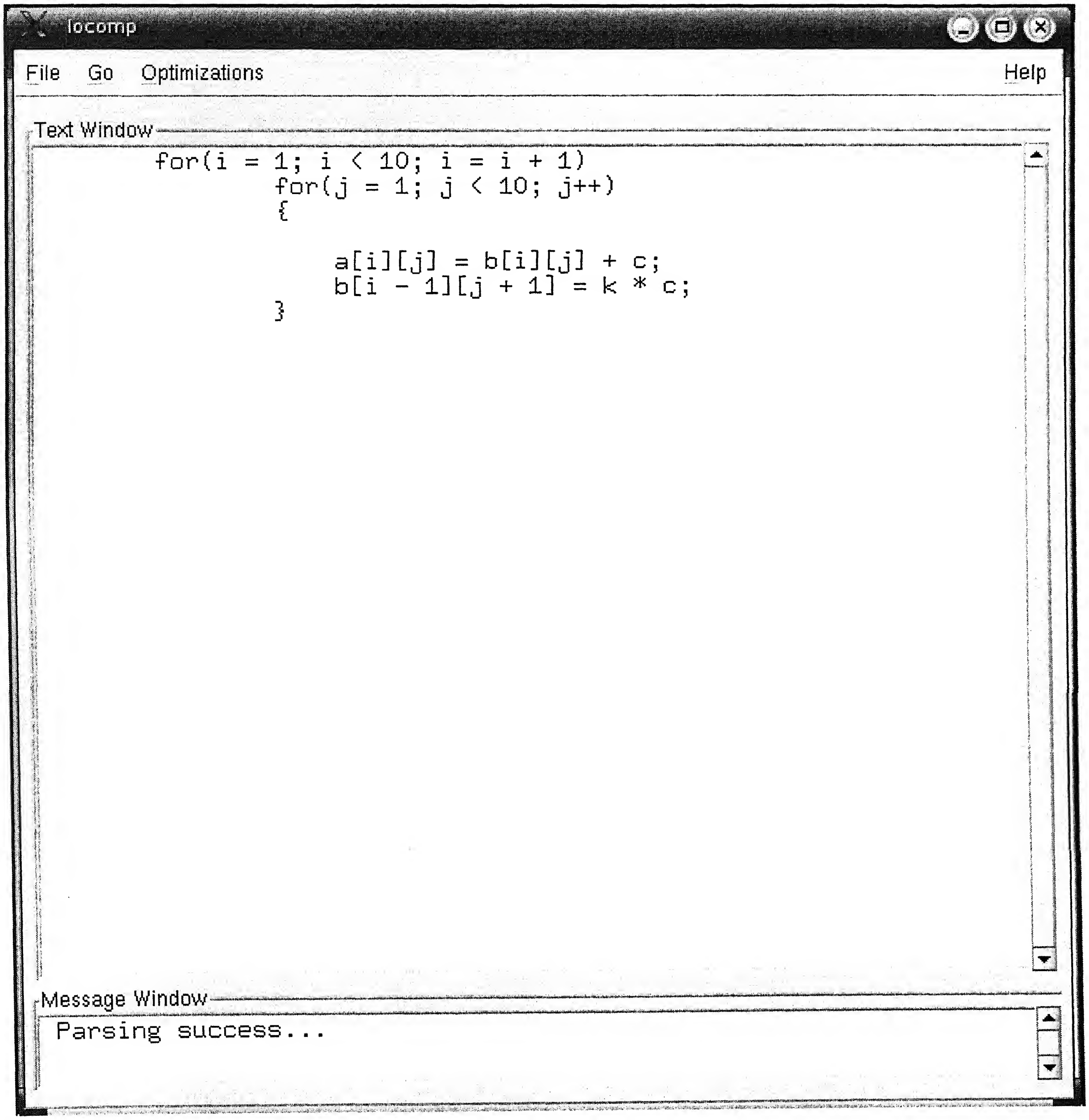
```
for(i = 1; i <= N; i++)
    sum += i;
```

4.6 User Interactive System

The user interaction with the compiler is through a graphical user interface (GUI). The GUI allows the user to apply loop optimizations on the *for* loops of the source program (*C* language). A snapshot of the graphical user interface of the tool is shown in figure 4.3.

The GUI consists of a *text window* and a *message window*. After the source program has been parsed successfully, the *text window* displays the first *for* loop in the source program. User can navigate to any one of the *for* loops in the source program through the use of commands in the *go* menu. The user can choose any one of the three optimizations (unswitching, unrolling and interchanging) from the *optimizations* menu to apply on the loop currently showing in the *text window*. The *message window* displays all the error and status messages. The *file* menu provides the commands for parsing and saving the source program.

The GUI is implemented using the GTK+ 2.0 [18]. GTK (gimp tool kit) is an application programming interface (API) for creating graphical user interfaces. The tool kit provides a concept of *widget*. Widgets are primitive graphical objects such as buttons, text boxes, menus, file selectors etc. A widget can be obtained by a library call. A graphical application is created by tailoring these widgets according to the application's requirement.



```
        for(i = 1; i < 10; i = i + 1)
                for(j = 1; j < 10; j++)
                {

                    a[i][j] = b[i][j] + c;
                    b[i - 1][j + 1] = k * c;
                }
```

```
Parsing success...
```

Figure 4.3: A snap shot of the GUI

Chapter 5

Conclusions and Future Work

We have developed a tool (source-to-source transformer), which can be used as a platform for experimenting with different kinds of loop optimizations and studying the effect of these optimizations. This helps researchers to learn the behavior of programs and to develop suitable algorithms for enhancing the capability of parallelizing compilers. It can also serve as a tool to demonstrate various loop optimizations in the teaching field. The tool abstracts the problem of manual transformations.

Future Work:

The main directions for future work are given below:

- The tool can be extended to include other optimizations like *loop fusion*, *loop distribution*, *loop tiling*, *loop skewing* etc. Extending the tool to include other optimizations is simple. A precondition checker and an optimizer for the particular loop optimization has to be provided to include that optimization in the tool.

- The code generation phase of the tool can be modified to generate code for vector and parallel machines. For generating parallel code, only the code generation phase of the loop needs to be changed.

- The tool can be extended to handle non-tightly nested loops

- We have not looked at pointer analysis and all the loops containing pointer operations are not considered as the candidates for optimization. Pointer analysis would help optimize a larger class of *for* loops in the *C* language.

Appendix A

Source Code Organization

This chapter gives the details of all files used in the implementation of the tool.

For building the tool, a *Makefile* is provided with the source code. The following command at the shell prompt builds the tool

```
$make
```

Before building the tool, the *GTK+* library and the *omega* library must be installed. The *GTK+* library is available at http://www.gtk.org. The *omega* library is available at http://www.cs.umd.edu/projects/omega.

The source code is organized into different modules. Each module's declarations are put in the header (.h) file and its implementation is put in the corresponding .c file.

1. absyn.h, absyn.c: contains routines for implementing the abstract syntax tree (AST) of the source *C* language.

2. checkdefined.h, checkdefined.c: contains routines for checking whether a variable is defined in a statement.

3. checkloop.h, checkloop.c: checks whether a *for* loop has index variables and increment expression.

4. clone.h, clone.c: contains supporting routines for obtaining a clone of a given statement or an expression in the abstract syntax representation.
5. codegen.h, codegen.c: contains routines for translating the abstract syntax tree (AST) into equivalent *C* language program.
6. ddtest.h, ddtest.c: contains useful routines for data dependence testing.
7. deallocmem.h, deallocmem.c: contains routines for deallocating dynamically created memory.
8. ecode.h: contains various error codes for reporting the errors to user.
9. gui.h, gui.c: contains the implementation of the *GUI* of the tool.
10. htable.h, htable.c: contains an implementation of hash table.
11. interchange.h, interchange.c: contains the precondition checker and the optimizer for loop interchanging.
12. loopback.h, loopback.c: contains the loop backs for all the widgets in the *GUI.*
13. loopdisp.h, loopdisp.c: contains routines for displaying a loop in the *text window* of the *GUI.*
14. looplist.h, looplist.c: contains a linked list representation of all the loops in the source program and supporting routines for navigating through the loops in the source program.
15. optimizations.h: contains declarations for all the loop optimizations.
16. parse.c: an interface between the front end and the *GUI.*
17. parser.y: The *YACC* specification file for parsing.
18. patch.c: contains routines for inserting a tree node into the abstract syntax tree.
19. scanner.lex: The *LEX* specification file for lexical analysis.

20. set.h, set.c: contains supporting routines for implementing a set.

21. symbol.h, symbol.c: contains supporting routines for representing symbols (strings).

22. table.h, table.c: implements the symbol table.

23. tool.c: the main driver program.

24. unroll.c: implements the precondition checker and the optimizer for the loop unrolling.

25. unswtich.c: implements the precondition checker and the optimizer for the loop unswtiching.

26. util.h, util.c: contains general supporting routines for use in the other modules.

Bibliography

[1] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[2] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, 1986.

[3] Apan Qasem Guohua Jin and John Mellor-Crummey. Improving Performance with Integrated Program Transformations. Technical Report TR03-419, Rice University, October 2003.

[4] Wei Li and Keshav Pingali. The Lambda Loop Transformation Toolkit (User's Reference Manual). Technical Report 511, University of Rochester, May 1994.

[5] U. Banerjee. *Data Dependence Analysis for Super Computing.* Kluwer Academic Publishers, Boston, MA, 1988.

[6] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM Press, 1991.

[7] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

[8] F.E. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30, 1972.

[9] Vivek Sarkar. Optimized unrolling of nested loops. In *Proceedings of the 14th international conference on Supercomputing*, pages 153–166. ACM Press, 2000.

[10] John R. Allen and Ken Kennedy. Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 233–246. ACM Press, 1984.

[11] C. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, 1988.

[12] Alfred V. Aho Ravi Sethi and Jeffrey D. Ullman. *Compilers: Principles,Techniques and Tools.* Addision Wesley Longman, 2000.

[13] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Narosha Publishing House, New Delhi, 1987.

[14] M.E. Lesk. Lex - A lexical analyzer generator. Technical Report 39, Bell Laboratories, 1975. UNIX programmars manual Vol 2.

[15] S.C. Johnson. Yacc - Yet Another Compiler Compiler. Technical Report 32, Bell Laboratories, 1975. UNIX programmars manual Vol 2.

[16] Andrew W. Apple. *Modern Compiler Implementation in C.* Cambridge University Press, 1998.

[17] William Pugh Evan Rosser Tatiana Shpeisman Wayne Kelly, Vadim Maslov and David Wonnacott. The Omega Library Version 1.1.0 Interface Guide. Technical report, University of Maryland, 1996. http://www.cs.umd.edu/projects/omega.

[18] GTK+ 2.0 Documentation. http://www.gtk.org.

[19] Steve S. Muchnich. *Advanced Compiler Design Implementation.* Harcourt Asia Pte. Ltd., 2000.

[20] Kleanthis Psarris. On exact data dependence analysis. In *Proceedings of the 6th international conference on Supercomputing*, pages 303–312. ACM Press, 1992.

[21] Wayne R. Cowell and Christopher P. Thompson. Transforming fortran do loops to improve performance on vector architectures. *ACM Trans. Math. Softw.*, 12(4):324–353, 1986.

[22] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 15–29. ACM Press, 1991.